Programming with methods and classes

Objective

- Explain class methods, class constants, class variables through the use of **static**
- Introduce the concepts of locality, scope, and nesting
- Introduce method overloading and overriding

Methods

- Instance method
  - Operates on a object (i.e., and *instance* of the class)

```
String s = new String("Help every cow reach its "
  + "potential!");
int n = s.length();
```

Instance method

- Class method
  - Service provided by a class and it is not associated with a particular object

```
String t = String.valueOf(n);
```

Class method

Data fields

- Instance variable and instance constants
  - Attribute of a particular object
  - Usually a variable

```
Point p = new Point(5, 5);
int px = p.x;
```

Instance variable

- Class variable and constants
  - Collective information that is not specific to individual objects of the class
  - Usually a constant

```
Color favoriteColor = Color.MAGENTA;
double favoriteNumber = MATH.PI - MATH.E;
```

Class constants

Task – Conversion.java

- Support conversion between English and metric values
  - 1 gallon = 3.785411784 liters
  - 1 mile = 1.609344 kilometers
  - d degrees Fahrenheit = (d – 32)/1.8 degrees Celsius
  - 1 ounce (avdp) = 28.349523125 grams
  - 1 acre = 0.0015625 square miles = 0.40468564 hectares

Conversion Implementation

```
public class Conversion {

  // conversion equivalencies
  private static final double
      LITERS_PER_GALLON = 3.785411784;


  private static final double
      KILOMETERS_PER_MILE = 1.609344;
  private static final double
      GRAMS_PER_OUNCE = 28.349523125;
  private static final double
      HECTARES_PER_ACRE = 0.40468564;
```

Conversion implementation

Modifier public indicates other classes can use the method

Modifier static indicates the method is a class method

```
public static double gallonsToLiters(double gallons) {
  return gallons * LITERS_PER_GALLON;
}
```

Observe there is no reference in the method to an attribute of an implicit Conversion object (i.e., a "this" object). This absence is a class method requirement. Class methods are invoked without respect to any particular object

Conversion Implementation

```
  // temperature conversions methods
  public static double fahrenheitToCelsius(double f) {
      return (f - 32) / 1.8;
  }

  public static double celsiusToFahrenheit(double c) {
      return 1.8 * c + 32;
  }

  // length conversions methods
  public static double kilometersToMiles(double km) {
      return km / KILOMETERS_PER_MILE;
  }
```

Conversion Implementation

```
// mass conversions methods
public static double litersToGallons(double liters) {
    return liters / LITERS_PER_GALLON;
}

public static double gallonsToLiters(double gallons) {
    return gallons * LITERS_PER_GALLON;
}

public static double gramsToOunces(double grams) {
    return grams / GRAMS_PER_OUNCE;
}

public static double ouncesToGrams(double ounces) {
    return ounces * GRAMS_PER_OUNCE;
}
```

Conversion Implementation

```
    // area conversions methods
    public static double hectaresToAcres(double hectares) {
        return hectares / HECTARES_PER_ACRE;
    }

    public static double acresToHectares(double acres) {
        return acres * HECTARES_PER_ACRE;
    }
}
```

Conversion use

Consider

```
double liters = 3.0;

double gallons = Conversion.litersToGallons(liters);

System.out.println(gallons + " gallons = " + liters
   + " liters");
```

Produces

```
Number of gallons: 3.0
3.00 gallons = 11.356235351999999 liters
```

A preferred Conversion use

Part of java.text

```
NumberFormat style = NumberFormat.getNumberInstance();

style.setMaximumFractionDigits(2);
style.setMinimumFractionDigits(2);

System.out.println(gallons + " gallons = "
   + style.format(liters) + " liters");
```

Rounds

```
3.0 gallons = 11.36 gallons
```

Practice: what's the output?

```
public class Demonstration {
   static private int counter = 0;
   public Demonstration() {
        ++counter;  // another construction has been performed
   }
   // main(): application entry point
   public static void main(String[] args) {
        Demonstration d1 = new Demonstration();
        Demonstration d2 = new Demonstration();
        Demonstration d3 = new Demonstration();
        System.out.println(Demonstration.counter);
   }
}
```

Question: what will happen if *counter* is an instance variable?

Method invocations and parameter passing

- **Actual parameters** provide information that is otherwise unavailable

```
double liters = 3.0;
double gallons = Conversion.litersToGallons(liters);
```

- When a method is invoked
  - Java sets aside *activation record* memory for that particular invocation
    - Activation record stores, among other things, the values of the **formal parameters** and local variables
  - Formal parameters initialized using the actual parameters
    - After initialization, the actual parameters and formal parameters are independent of each other
  - Flow of control is transferred temporarily to that method

Value parameter passing demonstration

```
public class Demo {
   public static double add(double x, double y) {
        double result = x + y;
        return result;
   }
   public static double multiply(double x, double y) {
        x = x * y;
        return x;
   }
   public static void main(String[] args) {
        double a = 8;
        double b = 11;
        double sum = add(a, b);
        System.out.println(a + " + " + b + " = " + sum);
        double product = multiply(a, b);
        System.out.println(a + " * " + b + " = " + product);
   }
}
```

Value parameter passing demonstration

Java Program Design

```
cmd:  javac Demo.java

cmd:  java Demo

8.0 + 11.0 = 19.0
8.0 * 11.0 = 88.0

cmd:
```

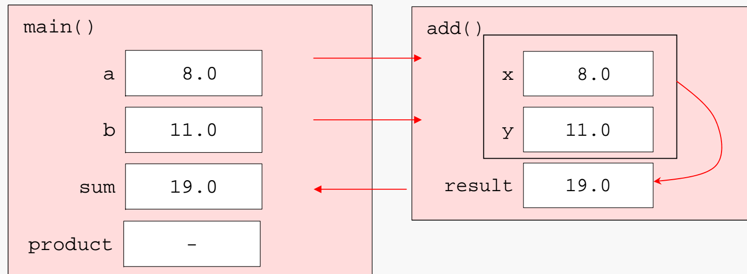
multiply() does not change the actual parameter a

Demo.java walkthrough

double sum = add(a, b);

Initial values of formal parameters come from the actual parameters

public static double add (double x, double y) {
double result = x + y
return result;
}

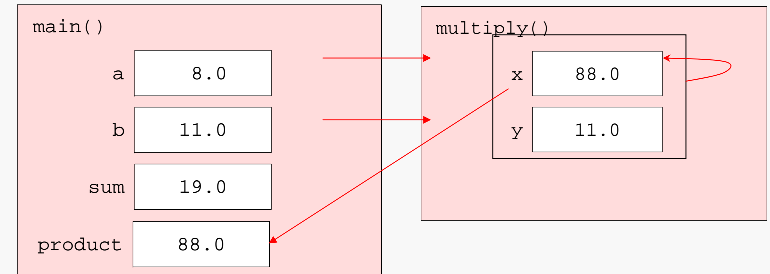


Demo.java walkthrough

double multiply = multiply(a, b);

Initial values of formal parameters come from the actual parameters

public static double multiply (double x, double y)
{
x = x + y
return x;
}



PassingReferences.java

```
public class PassingReferences {
   public static void f(Point v) {
       v = new Point(0, 0);
   }

   public static void g(Point v) {
       v.setLocation(0, 0);
   }

   public static void main(String[] args) {
       Point p = new Point(10, 10);
       System.out.println(p);

       f(p);
       System.out.println(p);

       g(p);
       System.out.println(p);
   }
}
```

PassingReferences.java run

Java Program Design

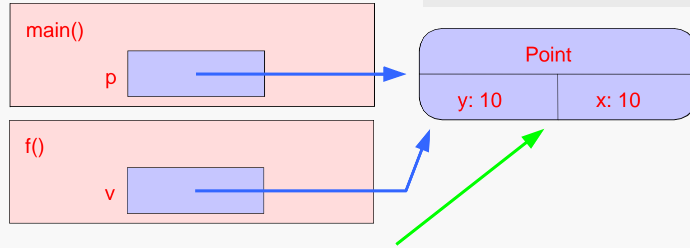
```
cmd:  javac PassingReferences.java

cmd:  java PassingReferences

java.awt.Point[x=10,y=10]
java.awt.Point[x=10,y=10]
java.awt.Point[x=0,y=0]

cmd:
```

g() can change the attributes of the object to which p refers

PassingReferences.java

```
public class PassingReferences {
    public static void f(Point v) {
        v = new Point(0, 0);
    }

    public static void g(Point v) {
        v.setLocation(0, 0);
    }

    public static void main(String[] args)
    {
        Point p = new Point(10, 10);
        System.out.println(p);

        f(p);
        System.out.println(p);

        g(p);
        System.out.println(p);
    }
}
```

```
public static void main(String[] args) {
    Point p = new Point(10, 10);
    System.out.println(p);

    f(p);
```



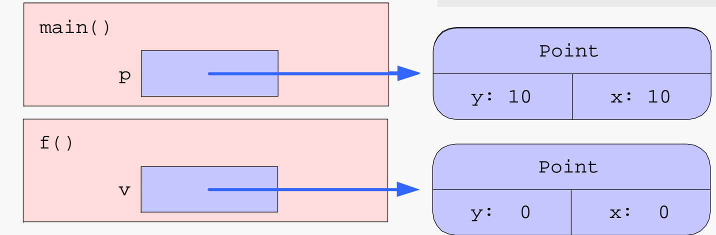
Method main()'s variable p and method f()'s formal parameter v have the same value, which is a reference to an object representing location (10, 10)

```
java.awt.Point[x=10,y=10]
```

PassingReferences.java


```
public class PassingReferences {
    public static void f(Point v) {
        v = new Point(0, 0);
    }

    public static void g(Point v) {
        v.setLocation(0, 0);
    }

    public static void main(String[] args)
    {
        Point p = new Point(10, 10);
        System.out.println(p);

        f(p);
        System.out.println(p);

        g(p);
        System.out.println(p);
    }
}
```

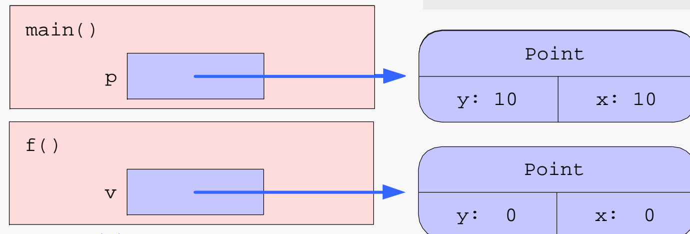

```
public static void f(Point v)
  {
    v = new Point(0, 0);
}
```



PassingReferences.java


```
public class PassingReferences {
    public static void f(Point v) {
        v = new Point(0, 0);
    }

    public static void g(Point v) {
        v.setLocation(0, 0);
    }

    public static void main(String[] args)
    {
        Point p = new Point(10, 10);
        System.out.println(p);

        f(p);
        System.out.println(p);

        g(p);
        System.out.println(p);
    }
}
```


```
public static void main(String[] args) {
    Point p = new Point(10, 10);
    System.out.println(p);

    f(p);
```



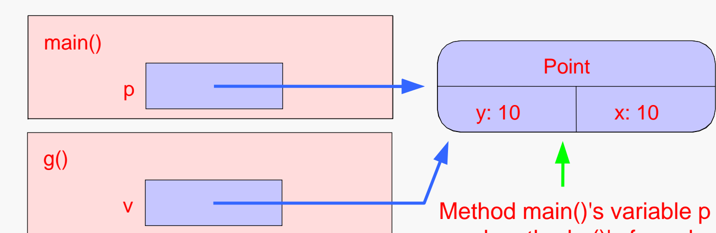

```
    System.out.println(p);
```

```
java.awt.Point[x=10,y=10]
java.awt.Point[x=10,y=10]
```

PassingReferences.java


```
public class PassingReferences {
    public static void f(Point v) {
        v = new Point(0, 0);
    }

    public static void g(Point v) {
        v.setLocation(0, 0);
    }

    public static void main(String[] args)
    {
        Point p = new Point(10, 10);
        System.out.println(p);

        f(p);
        System.out.println(p);

        g(p);
        System.out.println(p);
    }
}
```


```
public static void main(String[] args) {
    Point p = new Point(10, 10);
    System.out.println(p);

    f(p);
    System.out.println(p);

    g(p);
```



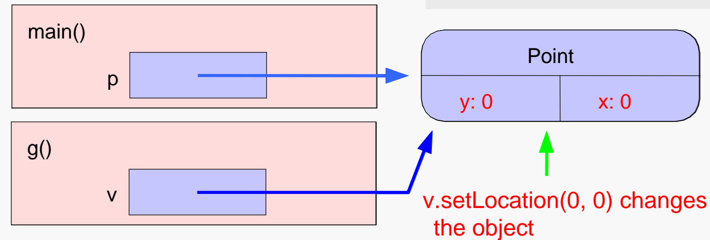
Method main()'s variable p and method g()'s formal parameter v have the same value, which is a reference to an object representing location (10, 10)

```
java.awt.Point[x=10,y=10]
java.awt.Point[x=10,y=10]
```

PassingReferences.java

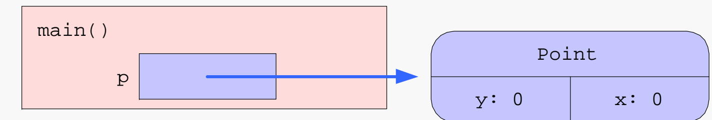
```
public static void g(Point v) {
    v.setLocation(0, 0);
}
```

```
public class PassingReferences {
        public static void f(Point v) {
                v = new Point(0, 0);
        }

        public static void g(Point v) {
                v.setLocation(0, 0);
        }

        public static void main(String[] args)
        {
                Point p = new Point(10, 10);
                System.out.println(p);

                f(p);
                System.out.println(p);

                g(p);
                System.out.println(p);
        }
}
```



PassingReferences.java

```
public static void main(String[] args) {
    Point p = new Point(10, 10);
    System.out.println(p);

    f(p);
    System.out.println(p);

    g(p);
```




```
    System.out.println(p);
```

```
java.awt.Point[x=10,y=10]
java.awt.Point[x=10,y=10]
java.awt.Point[x=0,y=0]
```

In Summary: the value of an actual parameter does not change with method invocation, however, if the actual parameter is a reference, then the object to which the actual parameter refers can be modified in a method invocation

What's wrong with the following program?

```
class Scope {
   public static void f(int a) {
       int b = 1;              // local definition
       System.out.println(a);  // print 10
       a = b;                  // update a
       System.out.println(a);  // print 1
   }

   public static void main(String[] args) {
       int i = 10;        // local definition
       f(i);              // invoking f() with i as parameter
       System.out.println(a);
       System.out.println(b);
   }
}
```

Variables a and b do not exist in the *scope* of method main()

Blocks and scope rules

- A block is a list of statements nested within braces
  - A method body is a block
  - A block can be placed anywhere a statement would be legal
    - A block contained within another block is a nested block
- A formal parameter is considered to be defined at the beginning of the method body
- A local variable can be used only in a statement or nested blocks that occurs after its definition
- An identifier name can be reused as long as the blocks containing the duplicate declarations are not nested one within the other
- Name reuse within a method is permitted as long as the reuse occurs in distinct blocks

Legal

```
class Scope2 {

  public static void f(int a) {
      System.out.println(a);
      a = 1;
      System.out.println(a);
  }

  public static void main(String[] args) {
      int a = 10;
      f(a);
      System.out.println(a);
  }

}
```

Legal but not recommended

```
public void g() {
  {
      int j = 1;                    // define j
      System.out.println(j);        // print 1
  }
  {
      int j = 10;                   // define a different j
      System.out.println(j);        // print 10
  }
  {
      char j = '@';                 // define a different j
      System.out.println(j);        // print '@'
  }
}
```

What's the output?

```
for (int i = 0; i < 3; ++i) {
  int j = 0;
  ++j;
  System.out.println(j);
}
```

- The scope of variable j is the body of the for loop
  - j is not in scope when ++i
  - j is not in scope when i < 3 are evaluated
  - j is redefined and re-initialized with each loop iteration

Task – Triple.java

- Represent objects with three integer attributes
- Constructor:
  - public Triple()
    - Constructs a default Triple value representing three zeros
  - public Triple(int a, int b, int c)
    - Constructs a representation of the values a, b, and c
- Assessor:
  - public int getValue(int i)
    - Returns the i-th element of the associated Triple
- Mutator:
  - public void setValue(int i, int value)
    - Sets the i-th element of the associated Triple to value

Task – Triple.java

- Facilitators:
  - public String toString()
    - Returns a textual representation of the associated Triple
  - public Object clone()
    - Returns a new Triple whose representation is the same as the associated Triple
  - public boolean equals(Object v)
    - Returns whether v is equivalent to the associated Triple

- These three methods are ***overrides*** of ***inherited*** methods from **Object**

Triple.java implementation

```
// Triple(): default constructor
public     Triple() {
    this  (0, 0, 0);
}
```

The new Triple object (the this object) is constructed by invoking the Triple constructor expecting three int values as actual parameters

```
public     Triple() {
    int    a  =  0;
    int    b =   0;
    int    c = 0;
    this   (a, b, c);
}
```

Illegal this() invocation. A this() invocation must begin its statement body

Triple.java implementation

```
// Triple(): specific constructor
public Triple(int a, int b, int c) {
   setValue(1, a);
   setValue(2, b);
   setValue(3, c);
}
// Triple(): specific constructor - alternative definition
public Triple(int a, int b, int c) {
   this.setValue(1, a);
   this.setValue(2, b);
   this.setValue(3, c);
}
```

Triple.java implementation

- Class Triple like every other Java class
  - Automatically an extension of the standard class **Object**
  - Class Object specifies some basic behaviors common to all objects
    - These behaviors are said to be inherited
  - Three of the inherited Object methods
    - toString()
    - clone()
    - equals()

Recommendation

- Classes should override (i.e., provide a class-specific implementation)
  - toString()
  - clone()
  - equals()
- By doing so, the programmer-expected behavior can be provided

```
System.out.println(p); // displays string version of
                       // object referenced by p
System.out.println(q); // displays string version of
                       // object referenced by q
```

Triple.java toString() implementation

```
public String toString() {
   int a = getValue(1);
   int b = getValue(2);
   int c = getValue(3);

   return "Triple[" + a + ", " + b + ", " + c + "]");
}
```

- Consider

```
Triple t1 = new Triple(10, 20, 30);
System.out.println(t1);

Triple t2 = new Triple(8, 88, 888);
System.out.println(t2);
```

- Produces

```
Triple[10, 20, 30]
Triple[8, 88, 888]
```

Otherwise: Object method toString() returns a string consisting of the class name appended with character '@' and the hash code of the object.

Triple.java clone() implementation

```
public Object clone() {
   int a = getValue(1);
   int b = getValue(2);
   int c = getValue(3);

   return new Triple(a, b, c);
}
```

Return type is Object (Every class is a specialized Object)

- Consider

```
Triple t1 = new Triple(9, 28, 29);
Triple t2 = (Triple) t1.clone();

System.out.println("t1 = " + t1);
System.out.println("t2 = " + t2);
```

Must cast!

- Produces

```
Triple[9, 28, 29]
Triple[9, 28, 29]
```

Triple.java equals() implementation

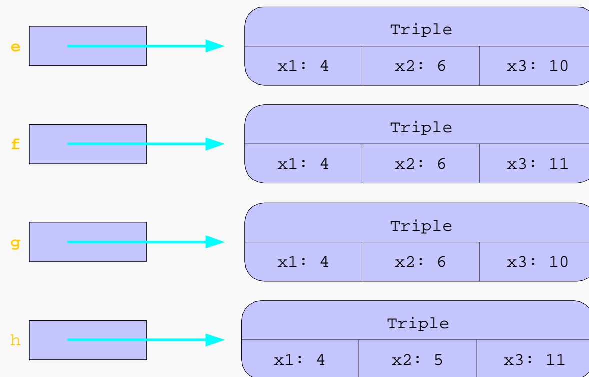
```
public boolean equals(Object v) {
   if (v instanceof Triple) {
      int a1 = getValue(1);
      int b1 = getValue(2);
      int c1 = getValue(3);

      Triple t = (Triple) v;
      int a2 = t.getValue(1);
      int b2 = t.getValue(2);
      int c2 = t.getValue(3);

      return (a1 == a2) && (b1 == b2) && (c1 == c2);
   }
   else {
      return false;
   }
}
```

Can't be equal unless it's a Triple

Compare corresponding attributes

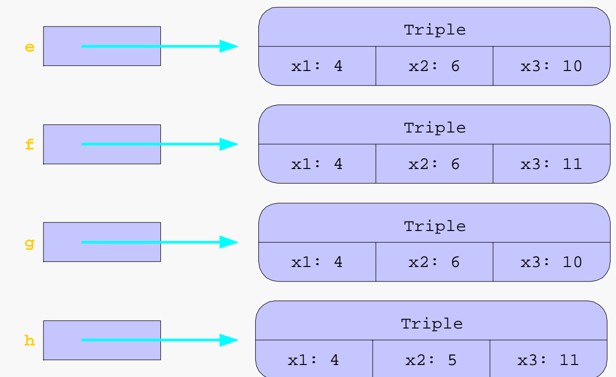
Triple.java equals()

```
Triple e = new Triple(4, 6, 10);
Triple f = new Triple(4, 6, 11);,
Triple g = new Triple(4, 6, 10);
Triple h = new Triple(4, 5, 11);
boolean flag1 = e.equals(f);
```



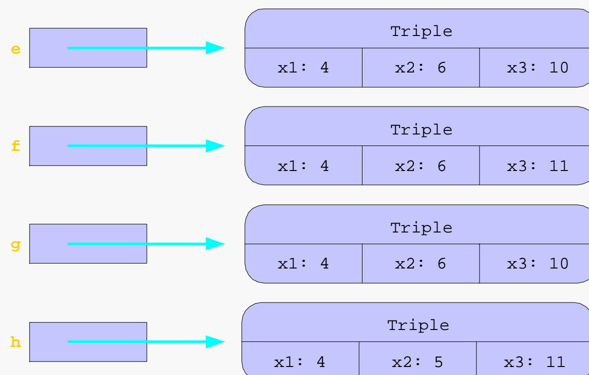
Triple.java equals()

```
Triple e = new Triple(4, 6, 10);
Triple f = new Triple(4, 6, 11);,
Triple g = new Triple(4, 6, 10);
Triple h = new Triple(4, 5, 11);
boolean flag2 = e.equals(g);
```



Triple.java equals()

```
Triple e = new Triple(4, 6, 10);
Triple f = new Triple(4, 6, 11);,
Triple g = new Triple(4, 6, 10);
Triple h = new Triple(4, 5, 11);
boolean flag3 = g.equals(h);
```



Overloading

- Have seen it often before with operators

```
int i = 11 + 28;
double x = 6.9 + 11.29;
String s = "April " + "June";
```

- Java also supports method overloading
  - Several methods can have the same name
  - Useful when we need to write methods that perform similar tasks but different parameter lists
  - Method name can be overloaded as long as its signature is different from the other methods of its class
    - Difference in the names, types, number, or order of the parameters

Legal

```
public static int min(int a, int b, int c) {
  return Math.min(a, Math.min(b, c));
}

public static int min(int a, int b, int c, int d) {
  return Math.min(a, min(b, c, d));
}
```

Legal

```
public static int power(int x, int n) {
  int result = 1;
  for (int i = 1; i <= n; ++i) {
      result *= x;
  }
  return result;
}

public static double power(double x, int n) {
  double result = 1;
  for (int i = 1; i <= n; ++i) {
      result *= x;
  }
  return result;
}
```

What's the output?

```
public static void f(int a, int b) {
  System.out.println(a + b);
}

public static void f(double a, double b) {
  System.out.println(a - b);
}

public static void main(String[] args) {
  int i = 19;
  double x = 54;

  f(i, x);
}
```

Assignment: due March 27, by noon

- Explain the following terms with examples: *class methods*, *class variables, override*, *overload*.
- 7.31 [Optional] (you must design the methods to be class methods)

For the following questions, you shouldn't get the results by running the program, you should be able to tell the results by just looking at the code segments, and also can explain why.

- 7.42 – 7.44
- 7.50 – 7.53
- 7.55
- 7.57-7.58